Open-Apple™

Releasing the power to everyone.

February 1986
Vol. 2, No. 1

ISSN 0885-4017
newstand price: $2.00
per page photocopy charge: $0.25

Apple unveils common sense

In mid-January, Apple held the 1986 Apple World Conference, its first extravaganza since last summer's reorganization (July issue, page 49). The conference brought together third-party hardware and software developers, dealers, educators, business customers, and user-group representatives to see Apple's newest products and to hear where Apple will take us next.

Apple unveiled a new Macintosh and a new LaserWriter, but the most impressive new item on display was corporate common sense—something long missing from Apple's portfolio. The new sensibility was evident in the plans of John Sculley, Apple's president, Del Yocam, Apple's executive vice-president for product operations, and Jean-Louis Gassee, Apple's vice-president for product development.

Sculley, Yocam, and Gassee appear to have abandoned Apple's traditional disregard for its installed customer base and have instead begun to recognize their customers as the foundation of Apple. Evidence of this change came in the form of a new attitude toward user groups, a new commitment to providing growth paths and upgrade programs for existing users, and a new policy of accepting trade-ins of older equipment.

In addition, Apple's years of stifling the Apple II family appear to be over. While no new Apple II products were introduced at the conference, Sculley, Yocam, and Gassee each made it clear that Apple recognizes the importance of the Apple II to the company's future.

According to Gassee, Apple has no intention of merging the Apple II and Macintosh product lines. Each will be developed into a family of systems with high- and low-end units and a universal line of peripherals. Gassee promised a continual flow of enhancements to these machines; he said there would be no more two- to three-year intervals between new products. He also promised to maintain the family architectures so that new machines would be able to do whatever older machines could do. And rather than talking about how the Apple II needs to look and feel like a Macintosh, Gassee said the Macintosh must be opened-up to "reincarnate the spirit of the Apple II in the 68000."

According to Yocam, Apple also intends to make a major effort to improve the speed and quality of the "service part of our product." Yocam acknowledged the importance of service to customers and mentioned VLSI designs (which use fewer chips), improved dealer systems, and maintenance of the quality of Apple's documentation as methods Apple will use to improve customer service.

Yocam also said that Apple sees a trend in personal computers towards their use as communication devices from their original role as computational devices. Yocam promised that Apple would make data communications an integrated part of the user interface. And he promised Apple would provide the ability to read data files on 3.5 inch disks without regard to what operating system had been used to write the files—watch out MS-DOS.

According to Sculley, enhancements to the Apple II will come in the areas of graphics, sound, and network possiblities. Sculley promised "real products that will bring network services to the classroom," linking students to teachers and teachers to the rest of the school, before the end of 1986. Sculley said the Apple II would "own" Christmas 1986, and that Apple planned to increase its international business to 35 per cent, up from 22 per cent of its business currently.

By adding our own common sense to the common sense of Sculley, Yocam, and Gassee, it's possible to peek under the veil covering the next Apple II. Because Gassee says Apple wants to develop product lines, expect the next Apple II to be a high-end addition to the Apple II family, rather than a replacement for either of the existing machines. Because he says Apple won't merge product lines, expect the new machine to be based on the 65816 chip rather than the Macintosh's 68000.

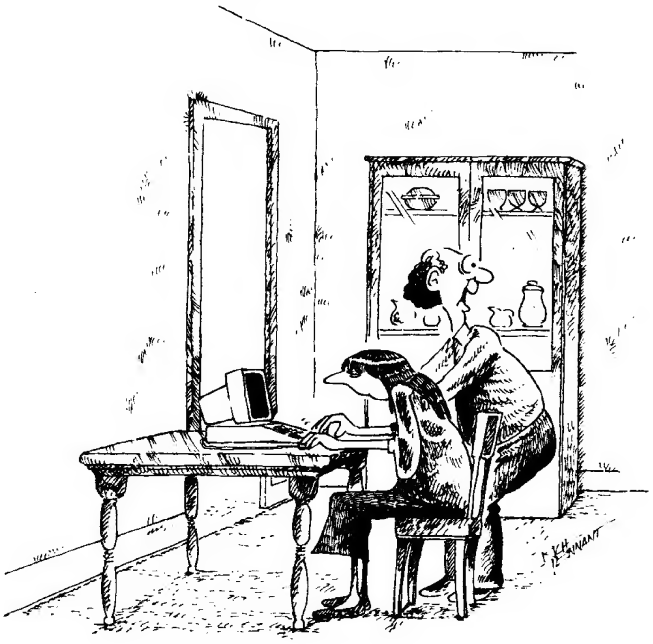
Because of Apple's emphasis on communications and its move toward a unified family of peripherals, expect to see an AppleTalk connection built into the next Apple II. Expect that the SCSI (say "skuzzy") high-speed interface used for hard drives on the new Macintosh will be available for the new II, although perhaps not built-in.

Because Apple considers itself a technological leader and says the Apple II is its machine in the consumer market, look for graphics and sound enhancements that put the II on par with Commodore's Amiga. Because Apple generally announces new products in January, May, or September, and because this new machine will get plenty of time for testing, development of documentation, and creation of programmer's tools, look for a September introduction.

In short, the Apple World Conference convinced me that the words "Apple II Forever" will carry a bit more weight nine short months from now. I hope I'm right.

Apple comes home to users

Apple has claimed for years that it is a "market-driven" company—that it takes its direction from its customers rather than from technology. In the very early months of Apple, when Steve Wozniak was designing the Apple II along the lines of what his user-group friends wanted, Apple was indeed "market-driven." Very quickly, however, the company began to follow the visions of Steve Jobs rather than the needs of its own customers.



"GATHER AROUND KIDS, YOUR MOTHER'S DOING WINDOWS!"

Wozniak's Apple II, the only Apple system built to fulfill customer visions, continues to sell well. The machines of Jobs have all been discontinued or drastically enhanced to meet market reality. This demonstrates why Apple, or any other company, should strive to be market-driven.

One of the clearest signs that Sculley, Yocam, and Gassee *mean it* this time when they say "market-driven" is Apple's new attitude toward user groups. The infant Apple Inc sprang forth from a user group, but quickly became an uncommunicative adolescent. Now, adolescence past, Apple Inc finally realizes that mom and dad weren't so stupid after all.

In his opening speech at the Apple World Conference, Sculley embraced user groups like this:

Among the people who will be carefully watching our development in both the Apple II and Macintosh product families will be Apple's expert users, representing over 200,000 members of computer clubs, people who know our products as well as we do ourselves.

We've always known that these groups of highly sophisticated users are some of the best evangelists we have. They're very influential in our sales, through word of mouth and after-sale support. And because we recognize just how valuable they are to Apple, I'm pleased to announce today the establishment of a support program between Apple and the nearly 600 Apple User Groups nationwide.

The two-way communication program we'll establish will strengthen the supportive relationships between user groups, dealers, and Apple. So to all of you out there who are part of Apple's user groups, I'm pleased our relationship will now be stronger.

In late November, Apple appointed Ellen Petry Leanse as User Group Evangelist and director of the user group program. Leanse quickly put together an excellent program for user groups at the Apple World Conference — including meetings with and speeches from Sculley, Gassee, Alan Kay, and Steve Wozniak — and attracted a hundred user group representatives to the conference. As Leanse continues to produce miracles like this, user groups can expect a greater exchange with Apple, both in terms of Apple providing support to user groups, and user groups helping Apple to gauge and respond to the needs and desires of the marketplace.

If Leanse hasn't contacted your group, I suggest you make yourself known to her — Ellen Petry Leanse, Apple Computer MS-23G, 20525 Mariani Ave, Cupertino, CA 95014.

Incidentally, I was one of the press people invited to hear Steve Wozniak's dinner speech to the user group representatives. Besides being a careful, creative engineer, Wozniak is also a wonderful speaker. He talked about Apple's early days and early connections with user groups in a rambling, humorous style that enraptured his audience. If you ever get a chance to hear him speak, don't miss it.

Bus School

The Magic of Peek and Poke

Among the most magical Applesoft commands are PEEK and POKE. Cryptic incantations that include these commands can create all sorts of enchantment. Novice and intermediate Applesoft programmers often cringe at PEEK and POKE, however, because there seems to be little rhyme or reason to how they work or what they do.

Even advanced programmers often use these commands by rote and by recipe. Just as the alchemist doesn't know why there are Smurfs in his broth, the programmer using a routine from a magazine or book may not know what POKE 49235,0 does or why. The most widely-available lists of interesting addresses to peek and poke — the Beagle Bros *Peeks, Pokes and Pointers* chart and William Luebbert's *What's Where in the Apple* — are often used like dusty alchemists' tomes. But knowing how PEEK and POKE make their magic isn't beyond the understanding of mere mortals.

Let's begin in a small Kansas town called Nortonville, where, many years ago, my grandfather ran a hardware store. His kind of hardware was nuts and bolts, lead pipe, brooms, and chicken wire. Nowadays, my grandfather would be surprised to learn, *hardware* is what computers are made of.

In 1959, eleven years after my grandfather died, Jack Kilby at Texas Instruments and Robert Noyce at Fairchild Semiconductor independently came up with the idea of an integrated circuit — a small square of silicon etched with thousands of electronic gizmos. During the 27 years since then, these silicon squares, mounted inside black plastic rectangles with metal legs that resemble insects, have radically changed the world.

Computers pre-date the silicon squares or *chips* of Kilby and Noyce by more than a decade. The 30-ton ENIAC, the world's first electronic digital computer, was dedicated in 1946, during my grandfather's lifetime. Real computers, however — the kind people like my daughter can use to search for *The Most Amazing Thing* — didn't appear until 1977; 31 years after ENIAC and 18 years after the first chips.

The two main "hardware" elements of any computer are a processor and memory. The development of real computers like the Apple II wasn't possible until these two elements were available on inexpensive chips. In the Apple II, the processor is tucked away on a single chip and, because of its relatively small size compared to ENIAC, is called a *micro*processor. The memory of an Apple II, while also small compared to ENIAC, requires several chips of two different types.

If an Apple II's chips were bees, the queen would be the microprocessor. It's one of the biggest chips in the computer and is the chip that does all the computing. The microprocessor in the original Apple II is known world-wide by its part number; it's called the "6502" (say sixty-five-oh-two). Nowadays Apples are built with a slightly enhanced version of this chip known as the "65C02" (say sixty-five-sea-oh-two). Herein we'll refer to both chips as "6502s."

If the Apple's microprocessor is a queen bee, then the Apple's memory chips make up her hive. There are thousands of honeycomb cells in Queen 6502's hive. Her power centers on her ability to use the honeycomb to store an unusual kind of honey — information.

Each cell in the honeycomb has an *address*. The addresses are more like Post Office box numbers than street addresses, since they just consist of a number, such as cell 99 or cell 14,945. The queen can communicate with each cell individually and directly. She doesn't have to call or walk past one cell to get to another. Because of this, the cells are said to be "randomly accessible."

Random access is a new concept to many people. Its opposite is *serial* access. A tape recorder, for example, is a serial-access device. The songs on a tape are lined up in a row — in a series. To hear any particular song you have to pass by other songs. There is always one song that is closest and one that is farthest away.

A real-life example of a random-access device is the telephone. You can call any other phone in the city directly. Your call never passes through or by a third phone. All phones appear equidistant. Likewise, Queen 6502 can access all the cells in her honeycomb independently and in equal, exceeding small (less than a millionth of a second) amounts of time.

What I have said about Queen 6502 so far is true no matter what brand of computer she reigns in (she's been used in several others besides the Apple II). Computers differ significantly, however, when we look at what the individual manufacturers have put in the cells of the honeycomb. This varies from computer to computer, and can even vary from instant to instant on the same computer, as we shall see later.

The memory honeycomb. Some cells can be used for temporary memory. The queen can store data in this type of cell and retrieve it later — as long as the cell is kept supplied with power. Turn the power off and the queen's data is lost forever. In the original Apple II, about 75 per cent of the cells were alloted to temporary memory.

Other cells are used for permanent memory. Data is placed in this type of cell when the computer is manufactured. It's as if the data was inserted and the cell was sealed with wax. Matthew Monitor and Dr. Basic live in cells like these. About a quarter of the original Apple II's cells contain memory of this type.

A few cells in the Apple II hold electronic switches for controlling devices connected to the computer. These cells are usually called "softswitches," because they contain switches that can be controlled with software. A few more cells are used as "ports" through which data can pass in and out of the

computer. Far less than one per cent of the cells in an Apple II are used for switches and ports.

Finally, some cells can contain nothing at all. This is unusual nowadays, but when the Apple II was first introduced, many were sold without a full complement of memory chips. Some of the chip sockets were simply empty because enough chips for 16,000 memory cells cost more in 1977 than chips for a million cells cost today.

If you counted the cells in Queen 6502's honeycomb, you would find exactly 65,536 of them. The very first cell has the address zero. The addresses go up sequentially to 65,535. The size of this honeycomb is fixed in the design of the 6502, as we shall see in a moment.

The cells that contain permanent data are called *ROM* or *read-only-memory*. Cells that are used for temporary data storage are called *RAM*, even though that doesn't make any sense. *RAM* stands for *random-access memory*. This doesn't distinguish it from ROM, since both types are accessed with random methods. RAM should have been called *WRM*, for write/read memory, but wasn't—probably because the engineers felt that giving a computer worms for brains was a disgusting idea.

The important thing to remember is that whenever the power company's transformer blows or you turn your computer off, data stored in RAM disappears forever. Data stored in ROM, on the other hand, can't be changed. Queen 6502 has the ability to write data into every cell in the honeycomb, but if she writes in a cell that contains ROM, the data won't stick. It's like writing on wax paper. Because of this limitation, programs stored in ROM are often called "firmware" rather than "software."

Each of the 65,536 memory cells holds a *byte* of memory. A byte is big enough to hold any number between zero and 255. If you have to explain to novices what a byte is, tell them that it's a unit of memory approximately equivalent to a single letter.

Also remember that novices feel they are drowning in alphabet soup when you talk about RAM and ROM. Especially since the size of RAMs and ROMs is measured in Ks. "K", of course, is widely used as an abbreviation for *kilo* or *thousand*. It doesn't help much, however, that as a measure of computer memory a K is 1,024 bytes rather than an even thousand.

Queen 6502's memory honeycomb of 65,536 bytes is exactly equal to 64K bytes of memory. In the standard Apple configuration, 48K of these are allocated for RAM and 16K for ROM. The ROM memory area includes 256 cells that are set aside for softswitches and data ports rather than memory cells.

A bit more magic. For all their enchanting properties, memory chips are really pretty simple. They consist of nothing but thousands of tiny little switches that can be either on or off. Each switch is known as a *bit* of computer memory. Each tiny switch can represent two numbers—zero and one. By combining these switches into groups, larger numbers can be represented.

For example, a group of eight switches can be set in 256 different on-off combinations. Technically speaking, a byte of memory is a group of eight switches. It takes eight 64K RAM chips, which are measured in bits, to make 64K of memory, which is measured in bytes. A "64K" memory chip houses 65,536 bits—a "64K" Apple has more than half a million of them.

When the microprocessor "reads" a byte of memory, what actually happens is that a combination of eight on-offs is copied from the designated switches in memory to a special set of switches inside the microprocessor. To accomplish this, eight "wires" or conductive paths running between the microprocessor and the memory byte are used. Switches that are "on" put one kind of signal on the data path. Switches that are "off" have a different signal. When reading, the microprocessor simply sets its internal switches to match the signals on the eight wires. For all this to work, each tiny switch in the memory honeycomb must be connected to one of the eight data paths.

In the world of electronics, conductive paths that have related functions and that are distributed together throughout a device are called a *bus*. In the Apple II, the eight lines the microprocessor uses to read the memory cells are called the *data bus*. The microprocessor also uses these same lines when writing to a memory cell—in this case, however, the signals flow the other way.

Before the IBM-PC was released, the power of a microprocessor was expressed in terms of the number of data lines the microprocessor used. The 6502, for example, has always been considered an "8-bit" microprocessor because it uses eight data lines.

Even though the 8088 microprocessor inside the IBM-PC also has just eight data lines, IBM's inventive sales engineers decided to proclaim it a "16-bit" chip. Their excuse was that—unlike the 6502—inside the 8088 itself 16 switches could be used at the same time for manipulating data. However, since the 8088 has to read and write those 16 bits eight bits at a time, a naked IBM is just slightly more powerful than a naked Apple II. The "16-bit" propaganda created other impressions, however, and worked well for IBM. (Apple retaliated much less successfully with the "32-bit" 68000 in the Macintosh, which is a true 16-bit chip as measured by the width of the data path.)

In addition to the data bus, 16 pins on the 6502 connect to a group of wires known as the *address bus*. This group of wires is used to designate which set of eight switches (i.e., which memory byte) the 6502 wants to read from or write to. A seventeenth wire is used to tell the memory chips whether the microprocessor wants to read or write.

The eight data lines, the 16 address lines, and the read/write line together make up most of the signals found on the 6502's motherboard socket. The remaining pins on the 6502 are for power, timing signals, and reset and interrupt signals.

As mentioned before, eight switches can be combined 256 possible ways. This is why the number 256 keeps showing up around computers. Note that the 1,024 bytes that make up 1K of memory is equal to 256 times 4. Queen 6502's memory honeycomb has 65,536 cells (256 times 256) because exactly this many different signal combinations are available using the 16 lines of the address bus.

How bank switching works. While Queen 6502's 16-line address bus can accomodate only 64K bytes of memory cells, extra chips that add much more than that are often used in Apples. An Apple IIc with the 3.5 UniDisk upgrade has 128K of RAM and 32K of ROM. Connecting all these bytes to the address bus requires some special electronic magic. The magical technique is known as *bank switching*.

Here's an historical example. The original 1977 model of Apple II had Steve Wozniak's Integer Basic sealed in its ROM chips. When the Apple II-Plus was released in 1980, the "plus" was that Applesoft Basic became the built-in language. Compatibility was an important issue in those days, so Apple provided several different ways Applesoft programs could be run on the older Integer Basic machines. One was to remove the older computer's ROM chips and replace them with ROMs containing Applesoft. Obviously, however, this particular technique could be a lot of trouble for users who didn't want to give up Integer Basic entirely.

So Apple came up with a device known as the *Applesoft ROM card*. The card included a set of Applesoft ROMs and was supposed to be plugged into slot zero. When the card was turned on, its Applesoft ROMs electronically replaced the Integer ROMs on the motherboard.

This can be likened to now-you-see-it, now-you-don't magic. It's as if a section of the memory honeycomb was suddenly removed and replaced with honeycomb from another hive. Queen 6502 doesn't see it happen and doesn't know the difference. Some of the things you can do with a little electricity are really amazing.

When an Applesoft ROM card is turned on, Applesoft appears in the memory cells where Integer Basic usually lives. When the card is turned off, Integer Basic suddenly reappears.

ROM cards were wonderful, but then Apple came up with a card that replaced the ROM chips with RAM. A RAM card allowed the 48K Apple's built-in ROM to be magically replaced with anything. Apple called its first RAM card a *language card*, because it allowed any Apple II to run either Applesoft or Integer Basic, as well as languages such as Pascal, Logo, and Fortran.

The bank switching technique adds one step to the method Queen 6502 uses with random-access memory. To use additional memory, a softswitch must be flipped that makes the alternate section of memory appear. To make the standard memory reappear, another switch must thrown. Since the 6502 itself doesn't know anything about these switches, only software can control which bank of memory is being used. Some software, such as Applesoft itself, doesn't know about bank switching and never uses it. Other software, such as RAMdisks and AppleWorks expansion programs, make extensive use of bank switching techniques.

Bank switching is a neat and useful feature of the Apple that allows tremendous flexibility and an unlimited amount of memory to be used by the 6502's "limited" 16-line address bus.

Memory Organization. As mentioned earlier, Queen 6502's 16-line address bus allows direct access to 65,536 memory cells. This is exactly equal to 256 times 256. It is common to consider this as 256 memory "pages" of 256 bytes each. These pages are numbered, as are most things in the computer world, starting with zero.

Novices often wonder why engineers didn't pick a "round" number to base computers on instead of the very square (16 times 16) 256. The answer, of course, is that we are forced to use unround numbers in computers by the nature of the switches—switches have just two fingers where we have ten. The roughness of computer numbers can be smoothed a bit, however, by grouping a byte's eight switches into two four-bit "nibbles."

Four switches can be combined in 16 possible ways. If we use 0 through 9 to designate the first ten combinations, and A through F to designate the last six, we can express any value that appears within a byte as a two-digit number. Zero becomes $00 (the dollar sign indicates we are using a 16-character, or "hexadecimal" numbering system) ten becomes $0A, 15 is $0F, 16 is $10, and 255 is $FF. This system is used all the time by assembly language programmers, because it makes the Apple's addresses "round." Poor old Applesoft doesn't know anything about it, however, and recognizes only decimal numbers.

By design, the 6502 uses several of its 256 memory pages in special ways. Machine language programs can access data on page zero in a sort of shorthand that is very quick. Page one is used by the 6502 as a "stack," which is special-purpose data storage area. Consequently, things work best when manufacturers put RAM in the memory cells for these two pages.

Likewise, when Queen 6502 sees a low voltage on her "reset" pin, she always stops what she is doing, grabs the address stored in bytes 252 ($FC) and 253 ($FD) of page 255 ($FF), and restarts execution at that address. Consequently, things work best when manufacturers put ROM in the memory cells for this page—which is the final one.

It is also helpful for computers to have all their RAM together in one place and all their ROM in another. Thus most computers that use the 6502, including the Apple II, have RAM in memory cells with low addresses and ROM in memory cells with high addresses.

When you turn an Apple II on, you'll find RAM in pages zero ($00) through 191 ($BF), softswitches in page 192 ($C0), and ROM in pages 193 ($C1) through 255 ($FF).

What light through yonder window breaks? When Steve Wozniak designed the Apple II, he also gave some other pages special qualities. Most importantly, his design causes the data stored in pages four through seven ($04-$07)to appear on your display screen. When you type a character and it appears on your screen, what has actually happened is that a number corresponding to your character was placed in a byte somewhere in this area; the Apple's video display hardware then takes notice and the character appears on your screen.

Wozniak also gave the Apple a second "text screen", on pages eight though eleven ($08-$0B). These pages are rarely used as a text display area, however. Instead, they are usually considered to be the first few pages of "free" RAM.

You tell the Apple's video hardware which of these pages you want displayed by throwing softswitches. Other softswitches tell the video hardware whether to interpret the data on these pages as text or as low-resolution graphics. Another softswitch commands the video hardware to display high-resolution graphics. In that case, the Apple uses the data on pages 32 ($20) through 63 ($3F)—high-res page 1—or pages 64 ($40) through 95 ($5F)—high-res page 2—as the source of the screen image.

As mentioned earlier, page 192 ($C0) is reserved for these softswitches. Pages 193 ($C1) through 255 ($FF) are reserved for ROM. The first 15 ROM pages, however, are the most interesting in the entire machine. What appears in this area of the memory honeycomb is ROM on the cards you insert into the Apple's slots. Page 193 ($C1) is reserved for ROM on the card in slot 1, page 194 ($C2) for ROM on the card in slot 2, and so on up to slot 7's page 199 ($C7). Pages 200 ($C8) through 207 ($CF) are shared, by means of bank switching, by all the slots. Since this scheme was used in the original Apple II, bank switching is primordal stuff in the Apple universe. Wozniak credits his friend Alan Baum with devising this section of the memory honeycomb.

On the original Apple II, pages 208 ($D0) through 223 ($DF) were connected to empty sockets on the motherboard. Integer Basic appeared in pages 224 ($E0) through 247 ($F7). The Apple II's operating system—Wozniak's Monitor—appeared in pages 248 ($F8) through 255 ($FF). When Applesoft appeared it used all the ROM area of Integer Basic plus the empty sockets—pages 208 ($D0) through 247 ($F7).

Peeking and poking at PEEK and POKE. Speaking of Applesoft, let's climb out of the Apple's hardware for a moment and look at the *software* at our disposal for probing the memory honeycomb. PEEK can be used to read the contents of any memory cell. Give the number of the cell you are interested in, in parentheses, after PEEK. To see what's in cell 32768 ($8000—page $80, byte $00), for example, do this:

```
PRINT PEEK (32768)
```

A statement like this will always return a decimal number between zero and 255, inclusive, since that is the entire range of what can fit in a single memory cell.

POKE can be used to deposit any value from zero through 255 in a memory cell. However, remember that POKE will have no effect if the cell you are poking at contains ROM. To poke 100 into cell 32768, for example, do this:

```
POKE 32768,100
```

If you try to PEEK or POKE at an address greater than 65,535, you will get an ILLEGAL QUANTITY ERROR, because you've fallen off the edge of the memory honeycomb.

Interestingly, however, PEEKS and POKES at addresses less than zero actually work. This is a heritage of Integer Basic. Although even preschoolers are taught that "numbers never stop", Integer Basic didn't know this. In the world according to Integer Basic, the final number was 32,767. In order to get at higher bytes, the Integer Basic PEEK and POKE commands accepted negative numbers. The scheme made byte zero equivalent to 65,536. Negative numbers caused PEEK and POKE to wrap around backwards from there. For example, PEEK(-1) accessed byte 65,535. PEEK(-32767) accessed byte 32,769. Interestingly, good old byte number 32,768 couldn't be probed with this system.

There are a number of interesting memory cells that came to be known by their negative number duing the days of Integer Basic. The three- and four-digit negative numbers of these cells are often easier to remember than the five-digit positive numbers that can be used with Applesoft (compare -151 to 65385, for example). Applesoft was designed to accept the negative numbers—all the way down to -65535, which will get you byte 1.

Note that both PEEK and POKE will accept variables:

```
ADR=32768 : VAL=100
PRINT PEEK(ADR) : POKE ADR,VAL
```

Both PEEK and POKE can also be used for flipping the softswitches on page 191 ($C0). However, some softswitches respond only to POKEs, some only to PEEKs, and some respond differently depending on whether they are peeked or poked. For some real-life examples, let's dip back into the hardware waters.

A closer look at Apple video. An elementary but easily overlooked characteristic of the Apple II is that all information that comes into the computer and all information that goes out passes through the cells in the memory honeycomb. Queen 6502 has **no** other connections to the outside world. The *only* things she can manipulate are the address and data buses. This scheme is known as *memory-mapped I/O*.

To get a feel for memory-mapped I/O, let's play with your Apple's 40-column text screen. In decimal, the very first byte on page four ($400) is number 1024. First try this program:

```
10 HOME
30 FOR CHR=0 TO 127
40 POKE 1024,CHR
60 NEXT
```

RUN the program and it will quickly POKE the values from 0 through 127 into byte 1024. As it does so, the character that each of these values represents to the video hardware will appear, very briefly, on your Apple's display.

The program is meant to convince you that what appears on your screen is a result of the values present in memory pages four through seven. Since our program puts all the values in same byte, however, you can't see much. Change line 40 and add lines 20 and 50 so that the program reads:

```
10 HOME
20 ADR=1024
30 FOR CHR=0 TO 127
40 POKE ADR,CHR
50 ADR=ADR+1
60 NEXT
```

RUN this version and you'll see three bars of characters appear on your screen. The bars will split your screen into upper, middle, and lower sections.

The characters actually displayed show you how the values from 0 through 127 appear on the screen.

Or do they? The screen is 40 characters wide and three lines are displayed. Three times 40 is 120. Characters 120 through 127 are missing! Change line 30 as shown and run the program again:

```
30 FOR CHR=0 TO 255
```

This time six lines or 240 characters are displayed. As the program runs, watch the order in which the six lines appear on your screen. Since we are poking values into sequential bytes, something very strange is happening.

Your display screen has 24 lines. Consider the top line number zero and the bottom line number 23. The first 40 bytes on page 4 appear as the top line on your display screen, the next 40 bytes as line 8, the next 40 bytes as line 16, and the next 8 bytes don't appear anywhere. The next 40 bytes appear on the screen's second line (line one), the next 40 on line 9, the next 40 on line 17, and the next eight nowhere.

The structure is Byzantine. The four memory pages are split into eight 128-byte segments, each of which holds a line in the top section of the screen, a line in the middle section, a line in the bottom section, and eight extra bytes. Novices find this organization confusing. Experts find this organization a beautiful example of Steve Wozniak's creativity and genius. The design allows a minimum amount of video hardware, a minimum amount of RAM, and the maximum number of characters that can be reliably displayed on a television set. Even the extra, undisplayed bytes, which are called the *screenholes*, end up being used by other parts of the Apple system.

Peek, poke, and tickle. To see a softswitch in action, type the following:

```
POKE 49232,0
```

Tickling byte 49232 with a PEEK or a POKE tells the Apple's video hardware to switch from a display of text to a display of low-resolution graphics. To switch back, tickle byte 49233. With this particular softswitch, it doesn't matter whether you use a PEEK or a POKE, or, if you POKE, what value you POKE into that byte. The switch just wants to be tickled. You'll find switching from low resolution to text can be tricky, because what you type will appear on the screen in low-resolution blocks rather than as text, but it can be done.

RUN our earlier program again—if you've been following along it should still be in memory—so that six lines of characters appear on your screen. Now tap return a few times so that the top line scrolls off the screen. What's happening here is that the software (or "firmware") built into the Apple Monitor is taking the values you placed in memory page 4 and moving them around so that the screen appears to scroll.

Our final group of values, which had been in line 17, are moved to line 16. Line 16 is moved to line 15, which is in memory page 7. It's all very complicated—but software makes it as simple as pressing the Return key.

While it's possible to manipulate the screen by poking values into pages 4 through 7, it's no way to run a railroad. I've shown you this to teach you about the power of PEEK and POKE and to show you how the Apple hardware works. Direct manipulation of the screen usually isn't a good idea. It's much better to use PRINT and let the Apple itself figure out where everything goes.

For example, switch to an 80-column display with a PR#3 and RUN our six-line SIX LINE program again. If each screen character uses up one byte, then an 80-column by 24-line screen will require 1,920 bytes of space. Our normal display area in pages 4 through 7, however, contains only 1,024 bytes, of which 960 are displayed and 64 are screenholes.

By turning on the 80-column screen and running our program, you'll demonstrate that, in 80-column mode, memory pages 4 through 7 are used for holding the characters in odd columns (assuming the first column is number zero). The characters for the even columns are stored elsewhere. Software that successfully POKEs messages directly on the screen in 40-column mode doesn't work too well in 80-column mode. Witness the NOT SELECTED message older Grappler-Plus printer interface cards sometimes poke onto the screen.

On the other hand, advanced software often finds it necessary to bypass the Apple's built-in firmware and use its own. *Apple Writer* has always done this for speed reasons. Programs that need to read information that's on the screen—perhaps a telephone number you want to dial, must also read the screen directly. The problem with such programs is that they must have special versions for each type of 80-column display device—and third-party hardware developers came up with several.

Peek, poke, and machine language. We've looked at how PEEK and POKE can be used to tickle softswitches and to change the memory area that is displayed on your screen. The other fundamental use of PEEK and POKE is to interact with machine language programs.

A machine language program tells Queen 6502 what to do. The instructions are stored in a series of memory bytes, either RAM or ROM. Typically the instructions tell her to read a byte, write a byte, compare two bytes, and so on—and they designate what byte to do it to. Reading, writing, comparing, adding, subtracting, and branching to other program segments pretty well sum up everything Queen 6502 can do. She can do them so quickly, however, that the wonderful result can be events like AppleWorks.

If a machine language program is in ROM, POKE can be used to modify it. This is what most of the POKEs given in ***Open-Apple*** are all about. By tweaking a RAM-based machine-language program such as DOS 3.3 here and there, you can make it do all kinds of enchanting stuff.

Another use for PEEK and POKE is to look at and change bytes that machine language programs are using for the storage of data. For example, the machine language program in the Monitor that automatically takes care of scrolling the display screen uses four bytes on page zero to change the size of the screen "window." You can make the data on the top four lines of the screen "permanent", for example, with a POKE 34,4. Byte 34 ($22) is a zero page location that the Monitor uses to remember the top edge of the text window (the top line is considered number zero). Byte 35 ($23) holds the bottom edge (but now the top line is considered number one), byte 32 ($20) holds the left edge, and byte 33 ($21) holds the window width. "The Wonderful World of Windows" in our June '85 issue (page 48), discusses these POKEs in detail.

Two-byte peeks and pokes. When the number a machine language program wants to remember is smaller than 256, as with our window dimensions, a single byte of storage will do. Often, however, it is necessary to remember larger numbers. Different programs use different formats for such numbers, depending on how many significant digits they must have and whether they must include a decimal point. One common kind of number stored by machine language programs, however, is a memory address. This is a two-byte (16-bit) number, and for reasons known only to microprocessor designers, the number is stored backwards from the way you would expect.

We know from earlier in this article that any address in the Apple II can be expressed as a two-byte number—one byte describes the memory "page" and the other the byte's position on that page. The machine-language Applesoft interpreter keeps the address of the lowest memory cell available to Applesoft programs in memory bytes 103 and 104 ($67-$68). Byte 103 holds the "byte" number and byte 104 holds the "page" number. The typical way to dig this address out of memory is like this:

```
PRINT PEEK(103) + PEEK(104)*256
2049
```

By multiplying the "page" byte by 256 and adding on the "byte" byte, we can translate the address stored in those two bytes. The answer is 2049 ($801), which is where the Applesoft programs you type in can generally be found.

By poking a different address into bytes 103 and 104, you can get Applesoft to put your program elsewhere. You might do this, for example, if you wanted to use "text page 2", which also uses byte 2049 and the 1,022 bytes that follow it.

To move the start of an Applesoft program to byte 3073 ($C01), just beyond the area needed by "text page 2", you need to POKE the value 3073 into bytes 103 and 104. There are several ways to do this. My favorite is to calculate the "page" first, then use the answer from that calculation to figure the byte, like this:

```
ADR=3073
POKE 104, ADR / 256 : POKE 103, ADR - (PEEK(104)*256)
```

(For this trick to work, you must also POKE a zero at ADR-1—otherwise Applesoft gets very confused. After making these POKEs, reLOAD your program, and it will load at the new position.)

To use PEEK and POKE effectively, of course, you need a magician's tome, such as those mentioned earlier, to know exactly what bytes control what. Now that you know why PEEK and POKE do what they do, however, you should be able to make your Apple II perform more enchanting magic.

Ask (or tell) Uncle DOS

DuoDisk numbers not serial

In your January '86 issue (page 98) you printed a warning to DuoDisk users about possible damage to disks that use certain copy protection schemes.

The number you published, however, is the part number of the DuoDisk printed circuit board located inside the drive, not the serial number of the affected units. The potential exists for problems to occur with units of almost any serial number.

Apple users who think they have a unit in need of upgrading should take the DuoDisk unit to an authorized Apple servicing dealer for a check. If the circuit board has not been upgraded, the dealer will do so at no cost to the user.

As with all software, the only way to positively avoid a catastrophic loss is to always keep backups of your disks.

Service Engineering
Apple Computer, Inc.

More track 0 crashes

I read with interest J. Ernest Cooper's letter on track 0 crashes in the January issue (page 103). I am a tech man in a school district that has in excess of 225 Apple II computers. This year we purchased over 70 of the new enhanced IIe machines.

In prior years we had some problems with losing data. We do have a program known state wide as being exceptional in quality of training for our staff and students. However, when you have 2,500 students K through 12 using computers, you have some human error involved. This year, however, with the addition of two word processing labs, and a phenomenal increase in the use of *AppleWorks* and *PFS:Write* by student and staff members, we have had a major increase in the number of data disk crashes, much more than the increased usage should warrant. Both *AppleWorks* and *PFS* use Track 0 for directories. The majority of the crashes occur during READ or WRITE activity.

Is our problem a problem with the enhanced IIe? I don't know. But Cooper's letter seems to point at a possible answer for the problem we have been having all this year. Does anyone else out there have similar problems cropping up?

As an addenda, your readers who have this problem might invest in *Bag of Tricks 2*, from Quality Software. It makes rebuilding damaged ProDOS directories a breeze.

Jim Aufderheide
New Ulm, Minn.

Do all these new IIes also have new 5-1/4 inch UniDisks? Or do they have the faulty DuoDisk drives? ***Open-Apple*** *correspondent Ken Kashmarek suggests the problem isn't the enhanced IIe but the enhanced disk drives. I've gotten a couple of other reports of track 0 crashes since publishing Cooper's letter. There's definitely something odd going on here. Keep those track 0 crash reports coming in.*

Other views on mouse/3.5

Ho-hum... two more super issues of ***Open-Apple.*** So what else is new?

Well, the UniDisk 3.5 ROM for one. It certainly is quite a collection of routines. Buried in the code is the option to read or write blocks of either 512 bytes (II family) or 524 bytes (Macintosh). Does that suggest any interesting avenues?

You ask why anyone would pay $560 for a UniDisk when they can get a 10MB hard drive for just a little more. One reason is portability. When a Mac/UniDisk conversion program becomes available (and I hear rumors now), you can stick a Mac disk in your shirt pocket and take it home to use on your IIe.

I disagree entirely with your comments on the mouse. I think there is a place for mouse technology and, as a good typist, I enjoy using the mouse when I'm word processing with Roger Wagner Publishing's *Mouse Write*. I suppose having the mouse present does slow execution somewhat, but if the program is written well, it should be transparent, or nearly so, to the user. Sure, I don't use the mouse for everything, there are cases where the alternative control commands work better for me. But on the other hand, when you spot an error a dozen lines up, a mouse sure beats entering a long string of keystrokes to get there and another to get back. I like it well enough that I even use it to create documents that have to be converted back to DOS 3.3 with Apple's @#(*%++& utilities. Try it, you'll like it!

Frank Andrews offered a neat trick to open a binary (or other) file in the January issue (page 99), but I'd like to be able to BLOAD a text file, as under ProDOS. In the same issue Paul Pagel mentioned my program "In the Dumps." An improved version of that routine, modified for ProDOS, appears on A.P.P.L.E.'s *ProZap* by Gary Charpentier.

Thanks for mentioning *On Three*. We have a UniDisk driver for the Apple III as well as great plans for 1986. There's a lot of life in the old gal yet, including the 65C802.

Val Golding
Tarzana, Calif.

Andrew's trick ***is*** *neat, but Weishaar messed it up pretty bad by giving the wrong address to poke. Those pokes at the top of the third column on page 99 should be to 42954 ($A7CA). The nine and the two were transposed in the original, both in line 10 and in line 40.*

Apple Pascal praised...

I'm sorry to hear that Apple Pascal reminds Uncle DOS of "a poke in the eye." Personally, ProDOS reminds me of a poke somewhere else, but that's another story.

I've been using Apple Pascal exclusively since it was released. For writing large commercial programs, it just can't be beat. The disk I/O is twice as fast as ProDOS, and the assembler is terrific.

In the January '86 ***Open-Apple*** (page 102), Stanley Cauthers asked about two bugs he found in Apple Pascal, both of which he was able to work around. I hardly think these bugs justify abandoning Apple Pascal, as you suggest.

The minor editor problem Cauthers mentioned happens to me about once a month. It's easy to recover from and I hardly notice it now. I've never run into the problem Cauthers describes with gradually running out of memory. I haven't used the 128K version of Pascal 1.2, so it may be a bug in that version.

Dynamic memory management with Pascal is a little tricky. I've discovered through painful trial and error that you shouldn't RESET or CLOSE any files in between a MARK and RELEASE, especially with Pascal 1.2.

The fact that Cauthers uses recursion in his program may be part of the problem. Basham's law of debugging recursive programs states that it is always harder to debug a recursive program than you expect, even after taking Basham's law into account.

As for getting support from Apple, I suggest Cauthers call Apple Developer Relations at 408-973-4897 to see about becoming a certified developer. Apple now has support service through MCI Mail, which should work a lot better than trying to call someone on the phone.

And if Uncle DOS expects anyone besides his mother to love him, he should watch who he pokes in the eye!

Bill Basham
Diversified Software Research
Farmington, Mich.

Okay, okay, sometimes I get a little carried away. I would be the first to admit that neither Applesoft nor assembly language, my personal favorites, are totally suitable for large commercial programs. But you have to admit that becoming a certified developer just to get support for an Apple product also seems a bit unsuitable.

Apple doesn't provide support for Applesoft either, but answers to Applesoft questions are much easier to obtain because so many more people understand the language and its operating systems. Few people, on the other hand, have attained a working knowledge of Apple Pascal and its operating system. I have recently jumped to the conclusion that this is not so much a problem with the Pascal language, which seems to be quite popular on other computers, but with the Apple Pascal operating system, which has so far prevented my own interest in Pascal from budding. Why do you think so few people have attained a working knowledge of Apple Pascal? How did you get past the prompt line?

...and damned

"A uniquely obstructive operating system..." This has to be the most elegant and incisive comment on Apple Pascal (January '86, page 102) ever to surface in any discussion of top-down, structured computer languages with their complex pseudo-opcodes, extravagant use of memory, and devious multiple boots versus good, old, clumsy, straightforward Basic with its prosaic line numbers, logical algorithms, user-friendly applications, and half a chance of returning control to the operator should it become necessary to swap horses in mid-stream for whatever valid reason.

One of the nasty surprises in store for novice Pascal users is the discovery that the operating system doesn't support direct operation of peripherals, such as a printer. All systematic configuration procedures to the contrary notwithstanding, the command TRANSFER APPLE3:ANYFILE.TEXT, #6:ANYFILE.TEXT doesn't even get a shrug from *my* printer, which functions perfectly from binary, Applesoft, integer, or text with PR#1. And that (*$L PRINTER:*) directive doesn't work either.

With Apple Pascal it appears that one must devise a machine language "driver" subroutine that gets installed somehow in SYSTEM.LIBRARY as a segment that is subsequently accessed and manipulated by SYSTEM.ATTACH with its own devious protocol and fussy syntax (whew!...all this just to get PR#1??? They're kidding!)

However, this particular cloud has a silver lining. In a copy of *Apple In Depth #2: All About Pascal* from A.P.P.L.E. Co-op there is a Pascal to DOS 3.3 conversion program called HUFFIN. It enables placement of a Pascal text file directly into memory from disk and printing it out by NORMAL and RATIONAL means that any reasonable apparatus can comprehend. Even my copy of Randy Hyde's P-SOURCE is so much excess baggage in this respect, along with a half-dozen other Pascal manuals in my collection that regularly collect dust.

In all fairness, an exception must be made for *Introduction to the UCSD P-System* by Charles Grant and Jon Butah (well written, no surprises, everything fits—except the printer thing), as well as the *Pascal Primer* by Fox and Waite, and *Apple Pascal, A Programming Guide* by Allen B. Tucker, Jr.

Even the Devil should have his day in court, if only to answer to charges of contending systems and general lack of cohesion and standardization of the Pascal hierarchy. But consider how long it took for the U.S. railroads to get together and agree on a standard rail gauge of 4 feet, 8-1/2 inches. THAT squabble took up most of the nineteenth century. Now, what's this about a Pascal standard, or any computer norm, for that matter?

Donald Ruch
Burbank, Calif.

MagiCalc to AppleWorks

I am a professional bookkeeper and I do word processing for the public. When I saw AppleWorks it sold me on buying a IIc. I have a lot of spreadsheets that I have built with MagiCalc and Ultraplan on my II-Plus. I now use AppleWorks constantly for word processing and databases, and would like to transfer my spreadsheets, using DIF files, to AppleWorks, too. I've tried to make the transfer unsuccessfully for hours. Could you print a step-by-step procedure for doing this?

My II-Plus is seeing less and less use these days because the IIc has more memory for my spreadsheets and because I use AppleWorks all the time. However, I still don't quite trust the IIc to hold up for years structurally (I get a lot of error messages with the built-in disk drive and my IIc monitor lasted for only four months before it stopped working) and I'm not fond of its closed design. If you had the choice, which machine would you invest in to buy more memory to run AppleWorks and create larger spreadsheets?

Marilyn Dresbach
Sonora, Calif.

First of all, forget about DIF files. DIF files are very handy for taking ***data*** *out of a spreadsheet so that it can be used by an Applesoft program or a data base manager, however, they are useless for moving stuff from one spreadsheet to another because they won't transfer* ***formulas*** *—just values.*

AppleWorks can directly load files that are in the VisiCalc storage format. All DOS 3.3-based spreadsheets that I know of use this format, however, I don't have any direct experience with MagiCalc or Ultraplan. But I'd bet that all you really have to do is convert your existing spreadsheet files from DOS 3.3 to ProDOS and load them into AppleWorks.

Here's the step-by-step: start up the System Utilities disk that came with your IIc. Use it to format a blank disk into the ProDOS format. Give this disk a simple name, "/a" would do fine. Now copy your existing spreadsheet files onto this disk, using the "copy files" option of the system utilities. The program will notice that you are copying from a DOS 3.3 disk to a ProDOS disk and make all necessary adjustments. Once the files are copied, start up AppleWorks and tell it you want to add some files to the desktop. Here's the tricky part—don't tell it the files are on a disk, tell it you want to make a new spreadsheet file. It will then ask you if you want to make the new file from scratch, from a DIF file, or from a VisiCalc file. Choose from a VisiCalc file. It will then ask you to type in the file's complete pathname. Type in the name of your ProDOS disk ("/a" if you've followed instructions), a second slash, and the name of your old file, e.g. "/a/ledger" (if the old filename had spaces in it, use periods in those positions, e.g. "/a/jan.ledger" for "jan ledger".)

Your spreadsheet will then load into AppleWorks. You may still encounter a few problems if your original spreadsheets use functions that AppleWorks doesn't support, such as LOG. AppleWorks handles most spreadsheets very nicely, however, and gives you a lot of expansion room. Make sure you like it before you start modifying your spreadsheets, however, because there is no way to move an AppleWorks spreadsheet back to your older programs.

If I owned a II-Plus and a IIc, had a business like yours, and didn't need any of the portability of the IIc, I'd attempt to work a trade of the pair of them for a used IIe. It makes a much stronger foundation to grow from.

A used IIe would give you the expandability and reliability you want and have the additional benefit of a better keyboard than either of the two computers you now have. The II-Plus keyboard, of course, suffers from a lack of keys. The IIc keyboard tends to stick a little. (If you prefer a heavier keyboard touch, incidentally, look for an early IIe—the kind where the keyboard letters are in the middle of the keys rather than in the upper-left corner as on the current models. I have one of those, and it's my favorite of the seven computer and two typewriter keyboards I've owned over the years.)

If you can afford to get a used IIe without giving up your IIc, it would provide you the benefit of a back-up computer in case something went wrong with your primary system.

There are also a few of us who find it efficient to use two computers at once. This is known as the one-person, two-computers philosophy. People who use more expensive computers can't afford this and instead have to resort to exceedingly complex "multi-tasking" software that allows two programs to run on the same machine at the same time—a much less preferable alternative.

Neat programs

In the January ***Open-Apple*** there were a couple of letters about changing various types of disk files into text format. There is a program on CompuServe in MAUG (DL3 library) called THE.EXECUTIVE (Copyright 1985, Living Legends Software) that claims to be able to change any type of file into a text file that can then be EXECed to restore it. I have tried this program and it works well and quickly, too.

Another program in MAUG DL3 is UNICOPY (Copyright 1985, Morgan Davis, Living Legends Software). It will copy an 800K UniDisk on a single drive with 8 disk swaps. Contrast this with the "duplicate a disk" option in the Apple II System Utilites version 2.1 that comes with the UniDisk 3.5. It takes a lot of disk swaps (forty, I think, but I got so frustrated I lost count).

Another neat "feature" of the new System Utilities occurs if you write protect the 3.5 inch disk it comes on. Booting the write protected disk causes the error WRITE PROTECTED, BREAK IN 406, and FILE(S) STILL OPEN to appear. This appears to be the ROM bug you mentioned on page 98. The STARTUP file on the System Utilities disk asks which language you want to work in (English, French, Italian, or German) when booted the first time. It creates a file with one character in it indicating the language selected. On following boots is uses this language but for some reason rewrites the file every time. This exposes the bug exactly as you described it.

Hugh McKay
Montreal, Quebec

Living Legends Software is a group of authors who distribute their programs as "freeware." You can get copies of their programs from CompuServe, try them out, and if you find something you want to add to your collection you send the author a check. Programs you don't want you can throw out.

AppleWorks page numbers

The is a response to Gary Morrison's AppleWorks page-numbering problems. I have had no problems printing page numbers, including a 159-page evaluation report. The AppleWorks word processor provides page numbering commands in the Printer Options Menu. To print page numbers:

```
1. Enter the Printer Options menu (OA-O)
2. Select Page Header (HE) or Page Footer (FO)
3. Select Page Number (PN) and enter the starting
     page number--you can skip this step if the
     document starts with page 1.
4. Select Print Page Number (PP)
5. Exit Printer Options Menu (ESC)
```

When printing a document, each page will have at the top (or bottom) a page number starting with the number entered in step 3.

Nicholas Cofsky Sky
Portland, Ore.

Here are some additional notes—when you do step 2, the message "-----Page Header" or "-----Page Footer" will appear on your screen. Whatever appears in the following line (you can type in anything you like) will print at the top (or bottom) of each ***following*** *page of your document.*

Step 4 puts a caret at the position the cursor was in when you pressed OA-O. Your page number will appear at this right-left position. You can also embed page numbers at any point within the text of a document with PP—even several times on the same page. The advantage of using a header or footer is that the page number will then appear automatically on every page. If you embed the page number in text (that is, anywhere but in the line immediately following a page header or page footer mark), you have to put it on every page manually. On page 79 of Robert Ericson's ***AppleWorks: Tips and Techniques****, there's an example of using the page number feature to create sequentially-numbered invoices.*

The PN command you mention in step 3 changes the number of the page the command appears in. For everything to work correctly, obviously, the PN command must appear ***before*** *the PP command on that page.*

Page numbering can also get messed up if you OA-P(rint) from "this page" or from the "cursor" rather than from the "beginning."

Incidentally, if you can't remember what a certain caret on your screen represents, place the cursor on it and look at the bottom of the screen. The line-column numbers are replaced with the function of the caret.

PFS to AppleWorks

Do you know of any way of converting *PFS:File* data bases into AppleWorks data bases? I have several billion bytes to transfer and it seems a waste of effort to re-type the information. Is there any way my Apple can do the work for me?

James Rusk
Garland, Texas

***Open-Apple** subscriber Jim Luther (5716 Forest, Kansas City, MO 64110) has written two programs for converting **PFS:File** and **PFS:Write** files into ProDOS text files that can be read by AppleWorks. The programs are limited—the **PFS:File** converter can take data only from the first page of a database record. If there are more than 30 fields on that page, or if any field has more than 78 characters, the excess is truncated, since AppleWorks would do that itself anyhow. **PFS:Write** files end up with a carriage return at the end of each line. Most of these carriage returns have to be deleted, since AppleWorks likes carriage returns only at the end of paragraphs. This program will also split long **PFS:Write** files into shorter segments.*

However, the programs definitely save a lot wear on your fingertips if you can live with these limitations. Luther will send you a disk with both programs on it for $20.

Open-Apple

[redacted]

is written, edited, published, and

Open-Apple
P.O. Box 7651
Overland Park, Kans. 66207 U.S.A.

ISSN 0885-4017. Published monthly since January 1985. Worldwide prices (in U.S. dollars; airmail delivery included at no additional charge): $24 for 1 year; $44 for 2 years; $60 for 3 years. All back issues are currently available for $2 each; seven or more from any single volume $14 (postpaid). Index mailed with the February issue. ***Open-Apple*** is available on disk for speech synthesizer users from Speech Enterprises, P.O. Box 7986, Houston, Texas 77270 (713-461-1666).

Source Mail: TCF238 CompuServe: 70120,202 Tele.: off hook

IIc color monitor interference

Recently I came across a problem with my Apple IIc that might be worth noting. I had the logic board upgraded for the UniDisk 3.5. About a month before I had bought a Color Monitor IIc, and everything had been working fine. After the upgrade I booted Infocom's *Enchanter* and the following message appeared, "INTERNAL ERROR 14. END OF STORY." I rebooted the disk and the drive whirred incessantly, making a horrid clattering noise intermittently. I tried other games, all of which I guess use DOS 3.3, and got pretty much the same result; with a few, the drive just turned without even recognizing there was a disk in the drive.

Two exceptions were Flight Simulator II and Lode Runner, which worked perfectly. When I took the computer to the dealer it worked flawlessly. He suggested I look for sources of "interference." I tried using my old monochrome monitor, and had no problem.

Hence, I guess there is something in the new logic board that is susceptible to interference by RF emissions by the color monitor, or the monitor produces unusual amounts of interference. Furthermore, this "interference" apparently affects programs depending on how they are copy-proteted, which might account for some of the programs working correctly.

I have tried booting the disks with the computer further away from the color monitor with only slightly better results. Pascal is also affected. Non-copy-protected DOS 3.3 disk and ProDOS disks are not affected.

Is there something I can do with the monitor to get the problem fixed? I have asked people who are knowledgeable, and so far the best idea I have heard is to get a thicker monitor cable. Also, what is INTERNAL ERROR 14?

Carl J. Schmidt
Morristown, N.J.

I recently purchased an Apple IIc color monitor and a UniDisk 3.5 disk drive. When I switched to the color monitor, I started to experience trouble booting and reading disks from the internal drive because of the RF interference coming through the front panel of the color monitor. I then tried to boot from the external floppy drive with a PR#7. To my surprise and delight an error message popped on the screen "AppleTalk Offline."

Investigating further, I have discovered that the new *Apple IIc Technical Reference Manual*, which includes the UniDisk 3.5 ROM listings, also contains full ROM listings for AppleTalk routines that are also built into the UniDisk 3.5 ROM upgrade. Apparently slot #7 will be used to access AppleTalk. Do you have any further information about this development?

Henry Landry
Andover, N.J.

Many people have had interference problems even with the Apple IIc monochrome monitor because the IIc's built-in disk drive and the monitor usually end up positioned very close to each other. I'm pretty sure the interference isn't RF (radio frequency) emissions, but magnetic emissions that confuse the disk drive head. The standard solution is to move the disk drive farther away from the monitor. Placing metal objects between the monitor and the drive could also divert enough magnetic energy to help.

Two interference reports in one month about the particular combination of the IIc color monitor and the 3.5 ROM upgrade may be meaningful, but I don't know how.

INTERNAL ERROR 14 must be an Infocom error message. It's not built-into the IIc anywhere, like AppleTalk is. The AppleTalk upgrade was apparently included in the 3.5 ROM upgrade so that yet another ROM upgrade wouldn't be required later. However, AppleTalk for the II family hasn't been released yet. I don't even know which connector on the IIc would use it.

ProDOS system file conversions

How can I save a ProDOS system file as a binary file? When I try to BLOAD a system file, I get a FILE TYPE MISMATCH error. If I use the dash command to get a system file into memory (which also causes it to execute, something I don't want to happen) and try to save it as a binary file by use of the syntax "BSAVE NEWFILE, A$2000", some disk activity occurs on the active drive and a PATH NOT FOUND error occurs. (Note that I am also trying to make use of the fact that it is allegedly unnecessary to specify the L parameter when BSAVING under ProDOS.)

Dan Strassberg
Arlington, Mass.

My copy of the Sider utility disk contains ProDOS 1.0.1. The Sider keeps the ProDOS kernel in a DOS 3.3 file so that it can be started up directly from DOS 3.3. I'd like to upgrade to ProDOS 1.1.1, but after all the horror stories I've read about the problems with CONVERT, I am more than a bit afraid to try converting it myself. If the CONVERT program will, in fact, convert the PRODOS file over to DOS 3.3 in a usable fastion, please tell us how to do it.

Tom Smith
Fort Vancouver, Wash.

To change the PRODOS kernel, or any other ProDOS system file, into a binary file, first BLOAD it using the T(ype) parameter like this:

```
BLOAD PRODOS, TSYS, A$2000
```

Next, BSAVE it using both the loading address and the length. You can get the length by looking at an 80-column CATALOG. It's the last number given for the file, and it's in decimal. Thus, for ProDOS 1.1.1:

```
BSAVE PRODOS.IMAGE, A$2000, L14848
```

*You do have to use the L(ength) parameter when saving a binary file, but only **the first time**. If the file already exists and you leave off the L parameter, Basic.system will use the length of the existing file. Notice that you get a PATH NOT FOUND error when you leave off the L with a new file rather than a SYNTAX ERROR, since Basic.system assumes your mistake was a bad path to the existing file rather than mistakenly leaving off the L.*

Once you have the ProDOS image in a binary file, you can use CONVERT or one of Apple's system utilities programs to convert it to DOS 3.3. I don't know of any problems with converting binary files. Nonetheless, I suggest you rename the original ProDOS image on your Sider and keep it around just in case.